Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation

Christine Choppy[1], Pascal Poizat[2], and Jean-Claude Royer[2]

[1] LIPN, Institut Galilée - Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr
[2] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
{Pascal.Poizat,Jean-Claude.Royer}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/

Abstract. Our KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views* [4,20]. In our unifying approach, views are used to describe the different aspects of a component (both internal and external structuring). We propose a semi-formal method with guidelines for the development of mixed systems, that helps the specifier providing means to structure the system in terms of communicating subcomponents and to describe the sequential components. While there is growing interest for having both textual and graphical notations for a given formalism, we introduce composition diagrams, a UML-inspired graphical notation for KORRIGAN, associated with the various steps of our method. We shall show how our method is applied to develop a KORRIGAN specification (both in textual and graphical notation) and illustrate this approach on a case study.

Keywords: formal specification, mixed specification, graphical notation, symbolic transition systems, KORRIGAN, UML

1 Introduction

The use of formal specifications is now widely accepted in software development to provide abstract, rigorous and complete descriptions of systems. Formal specifications are also essential to prove properties, to prototype the system and to generate tests.

In the last few years, the need for a separation of concerns with reference to static (data types) and dynamic aspects (behaviours, communication) appeared. This issue was addressed in approaches combining algebraic data types with other formalisms (*e.g.* LOTOS [17] with process algebras or SDL [7] with State/Transition Diagrams), and also more recently in approaches combining Z and process algebras (*e.g.* OZ-CSP [27] or CSP-OZ [8]). This is also reflected in

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 124–139, 2001.

object oriented analysis and design approaches such as UML [28] where static and dynamic aspects are dealt with by different diagrams (class diagrams, interaction diagrams, Statecharts). However, the (formal) links and consistency between the aspects are not defined, or trivial. This limits either the possibilities of reasoning on the whole component or the expressiveness of the formalism. Some approaches encompass both aspects within a single framework (*e.g.* LTS [24], rewriting logic [19] or TLA [18]). These "homogeneous" approaches ease the verification and the definition of consistency criteria for the integration of aspects, but at the cost of a loss of expressiveness for one of the aspects or a poor level of readability. Moreover, the definition of a method remains an important lack of most of these approaches.

The KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views*. Our approach aims at keeping advantage of the languages dedicated to both aspects (*i.e.* Symbolic Transition Systems for behaviours, algebraic specifications derived from these diagrams for data parts, and a simple temporal logic and axiom based glue for compositions) while providing an underlying unifying framework accompanied by an appropriate semantic model. Moreover, experience has shown that our formalism leads to expressive and abstract, yet readable specifications.

We propose, in this paper, a semi-formal method with guidelines for the development of mixed systems with KORRIGAN. This method helps the specifier providing means to structure the system in terms of communicating subcomponents and to describe the sequential components. While there is growing interest for having both textual and graphical notations for a given formalism (*e.g.* SDL and UML/XMI), we shall in the following introduce composition diagrams, a UML-inspired graphical notation for KORRIGAN, associated with the various steps of our method. We suggest to reuse some UML notation, but we also present some proper KORRIGAN graphic notations. Since our model is component based we have an approach which is rather different from UML on communications and concurrent aspects. Thus we will present specific notations to define the dynamic interface of a component, its communications with others and concurrency. We shall show how our method is applied to develop a KORRIGAN specification (both in textual and graphical notation) and illustrate this approach on a transit node case-study.

The paper is organized as follows. Section 2 presents the Transit Node case study. Section 3 gives an overview of KORRIGAN. It describes briefly the view model and details the associated specification method. Then, in Section 4 we present our UML-inspired notation diagrams for interfaces, compositions, behaviours, and communications. Finally, some related works are discussed in our conclusion.

2 The Transit Node Case Study

This case study was adapted within the VTT project from one defined in the RACE project 2039 (SPECS: Specification Environment for Communication

Software). It consists of a simple transit node where messages arrive, are routed, and leave the node (Fig. 1). We do not give the full informal specification text here but shortly describe what is needed in the context of this paper. The full presentation of the case study may be found in [20].

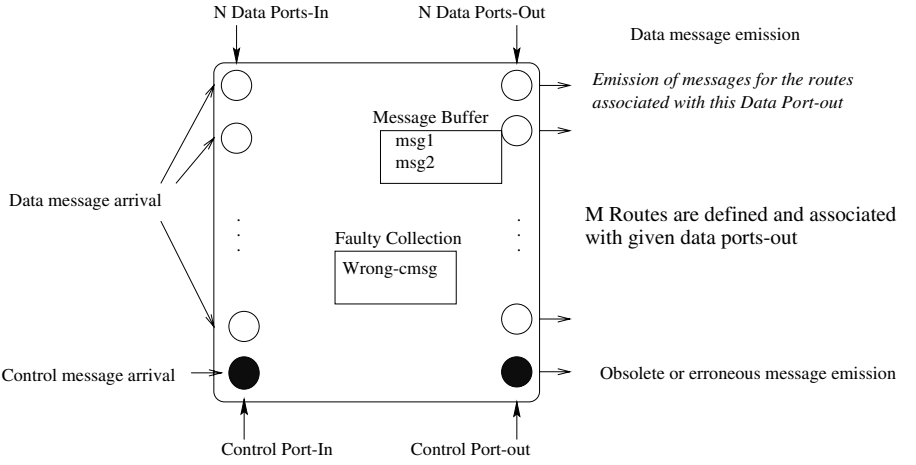

Fig. 1. Transit Node

The system to be specified consists of a transit node with one *Control Port-In* that receives control messages, one *Control Port-Out* that emits erroneous or obsolete messages, N *Data Ports-In* that receive data messages to be routed, N *Data Ports-Out* that emit data messages, and M *Routes* through. Each port is serialized, and all ports are concurrent to all others.

A data message is of the form *Route(m).Data*. The *Data Port-In* routes the message to any one of the open *Data Ports-Out* associated with the message route m where the message has to be buffered until the *Data Port-Out* can process it. When a message is erroneous (e.g. its route is not defined) or obsolete (its transit time is greater than a constant time T), it is eventually directed to a faulty collection.

The control messages modify the transit node configuration by enabling new data ports-in and out, or defining routes together with their associated data ports. The *Send-Faults* control message is used to route messages from the faulty collection to the *Control Port-Out* from which they will be eventually emitted.

3 Korrigan: A Formalism for Mixed Specification

In this Section, we will briefly present our model, the KORRIGAN specification language and the associated method.

3.1 Korrigan and the View Model

Our model [4,20] is based upon the structured specification of communicating components (with identifiers) by means of structures that we call *views* which are expressed in KORRIGAN, the associated formal language (Fig. 2).

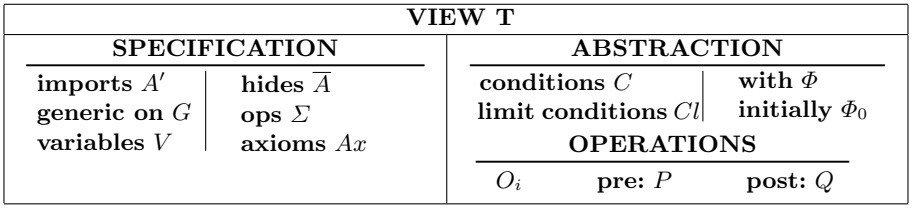

Fig. 2. KORRIGAN syntax (views)

Views use conditions to define an abstract point of view for components. These conditions are also used to define an inheritance relation for views. STS, *i.e. Symbolic Transition Systems*[1] are built using the conditions [21]. The main interest with these transition systems is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the dynamic and the static (algebraic) representation of a data type.

Views are used to describe in a structured and unifying way the different aspects of a component using "internal" and "external" structuring. We define an *Internal Structuring View* abstraction that expresses the fact that, in order to design a component, it is useful to be able to express it under its different aspects (here the static and dynamic aspects, with no exclusion of further aspects that may be identified later on). Another structuring level is achieved through the *External Structuring View* abstraction, expressing that a component may be composed of several subcomponents. Such a component may be either a global component (integrating different internal structuring views in an *Integration View*), or a composite component (*Composition View*). Integration views follow an *encapsulation principle*: the static aspect (*Static View*) may only be accessed through the dynamic aspect (*Dynamic View*) and its identifier (`Id`). The whole class diagram for the view model is given in Figure 3.

Components are "glued" altogether in external structuring views (Fig. 4) using both axioms and temporal logic formulas. This glue expresses a generalized form of synchronous product (for STS) and may be used to denote different concurrency modes and communication semantics. The δ component may be either `LOOSE`, `ALONE` or `KEEP` and is used in the operational semantics to express different concurrency modes (synchronous or asynchronous modes) and communication schemes. The `axioms` clause is used to link abstract guards that may

[1] Mainly transition systems with guarded transitions and open terms in states and transitions, see Figure 12 or [4].

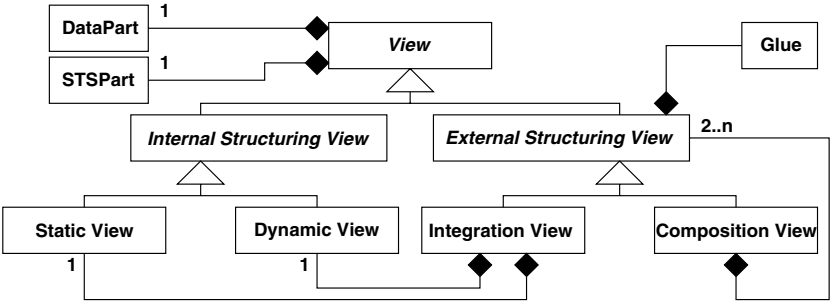

Fig. 3. Views class hierarchy (UML notation)

exist in components with operations defined in other components. The Φ and Φ_0 elements are state formulas expressing correct combinations of the components conditions (Φ) and initial ones (Φ_0). The Ψ element is a set of couples of transition formulas expressing what transitions have to be triggered at the same time (this expresses communication). The `COMPOSITION` clauses may use a syntactic sugar: the *range* operator (`i:[1..N]` or `i:[`e_1`, ... ,`e_n`]`), a bounded universal quantifier.

EXTERNAL STRUCTURING VIEW T	
SPECIFICATION	**COMPOSITION** δ
imports A' | **variables** V **generic on** G | **hides** $\overline{A}$	**is** $id_i : Obj_i\langle I_i\rangle$ | **axioms** Ax_Θ **with** Φ, Ψ **initially** Φ_0

Fig. 4. KORRIGAN syntax (compositions)

3.2 A Method for the Specification of Mixed Components

Methods are needed to help using formal specifications in a practical way. We propose a method for the development of mixed systems, that helps the specifier providing means to structure the system in terms of communicating subcomponents and to give the sequential components using a semi-automatic concurrent automata generation with associated algebraic data types. A previous version of our method [21] is described in terms of the agenda[2] concept [12,14]. The method presented here is refined and accompanied with the use of visual diagrams.

Our method mixes constraint-oriented, resource-oriented and state-oriented specification styles [29,30] and produces a modular description with a dynamic behaviour and its associated data type. Our method is composed of four steps (Fig. 5), with associated diagrams, for obtaining the specification.

[2] Agendas describe a list of activities for solving a task in software engineering, and are developed to provide guidance and support for the application of formal specification techniques.

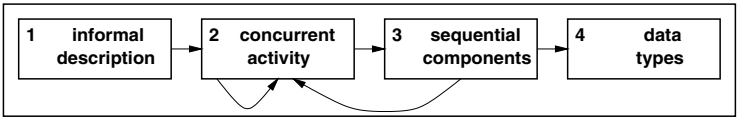

Fig. 5. Method Step Dependencies at the Overall Level

These steps correspond to:

- the **informal description** of the system to be specified. The aim of this first step is to sketch out the system characteristics (data, constraints and functionalities).
- the **concurrent activity** description. The idea is to define the *system architecture* in terms of a decomposition tree, and then to add *communication information* between the elements of this tree. This part on communication may be achieved after some basic components have been described (or reused). This step is composed of three sub-steps that may be iterated (Fig. 6).

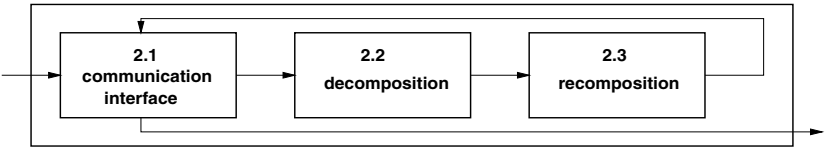

Fig. 6. Method Step Dependencies for the Concurrent Activity

In the first one (2.1), the *communication interface* of the component being described is given in term of an *interface diagram*.
Then, in a second step (2.2), the component is decomposed into concurrent (possibly communicating) subcomponents using *composition diagrams*. Once a sequential component is obtained, a further decomposition step is applied to reflect the fact that components are indeed the integration of different aspects (static and dynamic). Therefore, we have *concurrent decompositions* (described by *concurrent composition diagrams*) and *integration decompositions* (described by *integration composition diagrams*), *i.e.* separation of aspects steps.
In the third step (2.3), the different composition diagrams (either integration or concurrent composition diagrams) are completed with the communications that take place between them. This can be done by reusing communication patterns. When they are completed with communication information, the composition diagrams are called *communication diagrams*. Note that in our KORRIGAN framework, both concurrency and integration are specified in a unified way.

Table 1 gives the correspondence between the concurrent activity sub-steps, the corresponding diagrams, and the corresponding KORRIGAN view structures. All the diagrams will be presented in the next Section.

Table 1. Method steps, diagrams and KORRIGAN view structures

step	diagram	Korrigan
communication interface	interface (e.g. Fig. 8)	Σ in Internal Structuring Views `SPECIFICATION` part
decomposition	composition (e.g. Fig. 9)	External Structuring View (partial: `imports` and `is` clauses)
recomposition	communication (e.g. Fig. 14)	External Structuring View (full)

- the **sequential component** descriptions. The term "condition" refers to preconditions required for a communication to take place, and also to conditions that affect the behaviour when a communication takes place. The operations are defined in terms of pre and postconditions over these conditions. These concepts correspond to our formal language (see Fig. 2) but gives a general method for a wider set of mixed specification languages (it has been applied to LOTOS and SDL). A guarded automaton is then progressively and rigorously built from the conditions. Type information and operation preconditions are used to define the automaton states and transitions. A dynamic behaviour (described by a *behavioural diagram*) may then be computed from the automaton using some standard patterns.
- the **data type** (functional) specifications. The last improvement is the assisted computation of the functional parts (in KORRIGAN, the data type parts of views). Our method reuses a technique [2] which allows one to get an abstract data type from an automaton. This technique extracts a signature and generators from the automaton. Furthermore, the automaton drives the axiom writing so that the specifier has only to provide the axioms right hand sides.

After a preliminary presentation of our UML-inspired notation, we introduce the various diagrams supporting our formal specification method illustrated with the Transit Node case study.

4 A UML-Inspired Graphical Notation

UML [28] is a notation to be used for object-oriented analysis and design. Since it is very expressive (it has 11 different diagram types), and its (informal) semantics is unclear in some cases [23], it is common to restrict oneself to a subset of it, but there are also proposals to modify/extend it [11,16].
We think that, in complement to the theoretical approach that tries to formalize

the UML [22,9], an interesting and more pragmatic approach is to reuse existing well-accepted semi-formal notations and use them as a graphical means to improve the readability of formal languages and concepts. In order to be close to the UML, we select a subset of UML that is relevant to illustrate concepts of our approach. We also extend/modify it when needed. For instance (Fig. 8), we use a simple class diagram together with informations related to the possible communications (its interface) of the component. The interface symbols [3] we use are described in Figure 7.

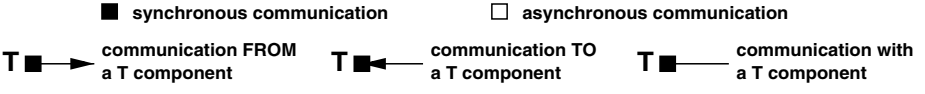

Fig. 7. Communication Interfaces

There exist different kinds of compositions within the set-theoretic or the object-oriented framework. We restrict ourselves to the strong composition of critical systems and distributed applications, that is composition with dependence, exclusivity and predominance. This can be compared with the strong composition in UML (black diamond), therefore, we use this UML symbol to represent it (Fig. 9). However, our approach is more component-oriented than UML since we explicitly address communication issues in interfaces (Fig. 8) and concurrency in communication diagrams (Fig. 15) whereas in UML they are embedded within Statecharts.

4.1 Interface and Composition Diagrams

We introduce composition diagrams to decompose a component into subcomponents. We denote by (de)composition both the separation/integration of the different aspects of a component (static and dynamic), and the (de)composition into concurrent communicating subcomponents. The KORRIGAN model enables us to describe both, in a unifying way, using specific External Structuring Views, respectively Integration and Composition Views.

Interface Diagrams. Following our method, at an abstract level of description, the transit node may be described using its data and its functionalities.

We may now give the description of the transit node at the most abstract level (Fig. 8). It has four functionalities. Its data are composed of three lists. The transit node is parameterized by `N`, the maximal number of ports within.

[3] Note that KORRIGAN has no support for asynchronous communication, hence it is achieved through buffers.

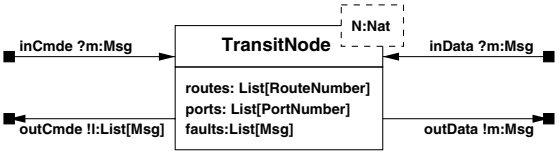

Fig. 8. `TransitNode` Interface Diagram

Composition Diagrams. Following our method, we decompose the transit node into control ports and data ports using two views: `ControlPorts` and `DataPorts`. We then distribute the transit node functionalities and data in them. Finally, we name the subcomponents of the transit node (`control` and `data`). Figure 9 represents the first level of decomposition of the transit node.

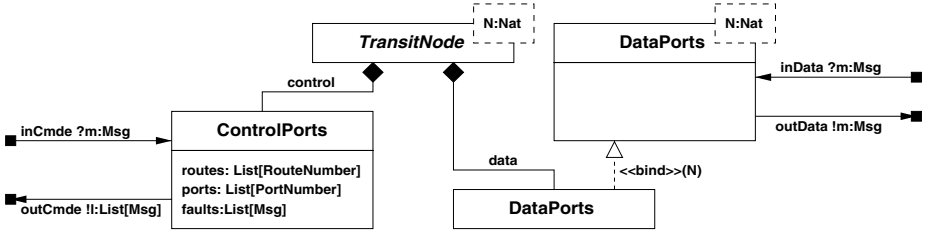

Fig. 9. `TransitNode` Composition Diagram

At this step, we may retrieve a partial KORRIGAN specification for the transit node using its graphical representation. Such a specification would be partial because the component views it uses (`imports` clause) may have not been defined yet. The communications between the subcomponents (that is the "glue" between them) will be specified in a latter step but this does not always implies that the super-component is partial because there may not be any communications at all between two subcomponents.

Applying the same decomposition process on the `DataPorts` view, we obtain the Figure 10 diagram. Note here that its subcomponents make use of the range operator to express a set of identifiers. The `OutputDataPort` has a buffer to deal with its serialization constraints. We do not treat the `ControlPorts` here by lack of place, see [20].

Integration Composition Diagrams. As mentioned above, in KORRIGAN the integration of the different aspects of components within a global one is also a kind of composition. However, to distinguish between integration and concurrent composition, we use integration diagrams: composition diagrams where the names of the integration components are put into gray boxes.

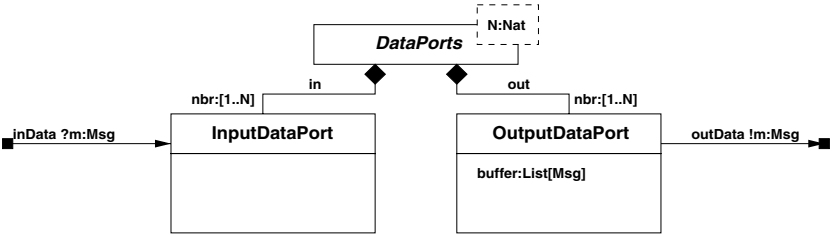

Fig. 10. `DataPorts` Composition Diagram

In the integration diagrams, the data that were in previous composition diagrams are transformed into corresponding static aspects. It is a matter of design to choose if each data will have its corresponding static aspect view, or if several may be incorporated into a single static aspect view. This last solution complicates the description of the communication between aspects and diminishes the reusability level of the components. It is important to note that when a component has no data, it is integrated with a trivial (`Null`) static part. See [20] for the case study diagrams.

4.2 Behavioural Diagrams

Basic components are specified using views. Such a view may be given as a triple (`SPECIFICATION`, `ABSTRACTION` and `OPERATIONS` parts), or using the STS derivation principles [21], as a couple (`SPECIFICATION` part and a STS). These STSs may be related to Statecharts ([13], or UML ones) but for some differences:

- STSs are simpler (but less expressive) than Statecharts;
- STSs model sequential components (concurrency is done through external structuring and the computation of a structured STS from subcomponents STSs [4]);
- STSs are built using conditions which enable one to semi-automatically derive them from requirements;
- STSs may be seen as a graphical representation of an abstract interpretation of an algebraic data type [2].

We give in Figure 11 a part of the `InputDataPort` KORRIGAN specification, and in Figure 12 the corresponding behavioural diagram.

In presence of generic components, we may use instantiation diagrams to relate concrete components to their generic parent. Such views are reusable. Generally, the static views are sets, lists or buffers, *i.e.* the description in dynamic terms of the inputs and outputs of a storage element. In Figure 13, the instantiation process is used for the different static views describing lists.

There are also inheritance diagrams in our model [20].

DYNAMIC VIEW `InputDataPort`

SPECIFICATION

imports
Msg, RouteNumber, PortNumber
variables fc: FaultyCollection
ops
enable
FROM InputControlPort
inData ?m:Msg
FROM InputControlPort
askRoute !r:RouteNumber
TO InputControlPort
replyRoute ?l:List[PortNumber]
FROM InputControlPort
wrongRoute !m:Msg
TO FaultyCollection
correct !m:Msg
TO OutputDataPort
axioms see [20]

ABSTRACTION

conditions
enable, received, asked, replied, routeErr
with
replied ⇒ enabled
asked ⇒ received
replied ⇒ asked
routeErr ⇒ replied
initially ¬ enabled

OPERATIONS

enable
pre: true
post: enable' : true
inData
pre: enable ∧ ¬ received
post: received' : true
...

Fig. 11. `InputDataPort` in KORRIGAN

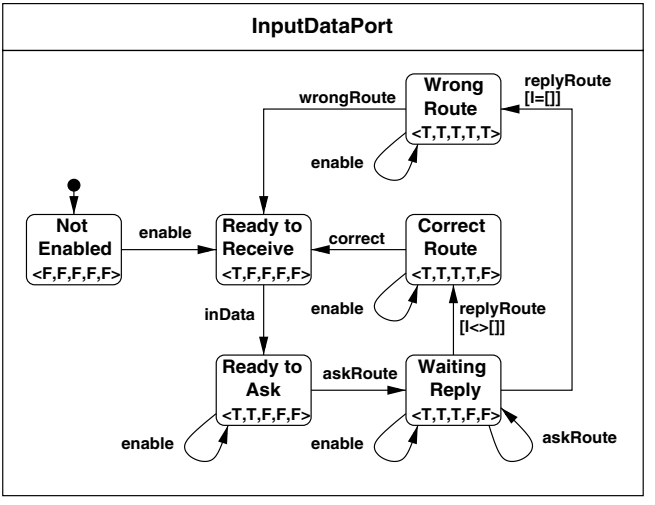

Fig. 12. `InputDataPort` Behavioural Diagram (STS)

4.3 Communication Diagrams

The communication diagrams are used to complement the composition diagrams with the inter-component communication and concurrency schemes. They use a graphical notation of the KORRIGAN glue rules (`COMPOSITION` parts in external structuring views, Fig. 4).

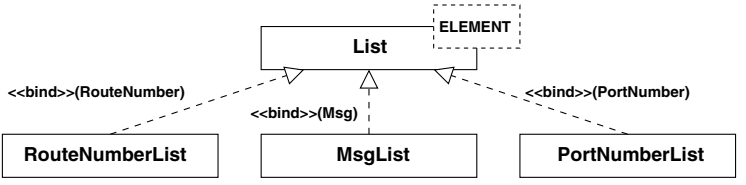

Fig. 13. `List` Instantiation Diagram

The axiomatic part of the glue (the `axioms` clause), the state temporal formulas (Φ and Φ_0), and the concurrency mode (δ) are put in the aggregating component (*i.e.* `DataPorts` for the `InputDataPort` and `OutputDataPort` views) as shown in Figure 14. Here there are no glue axioms.

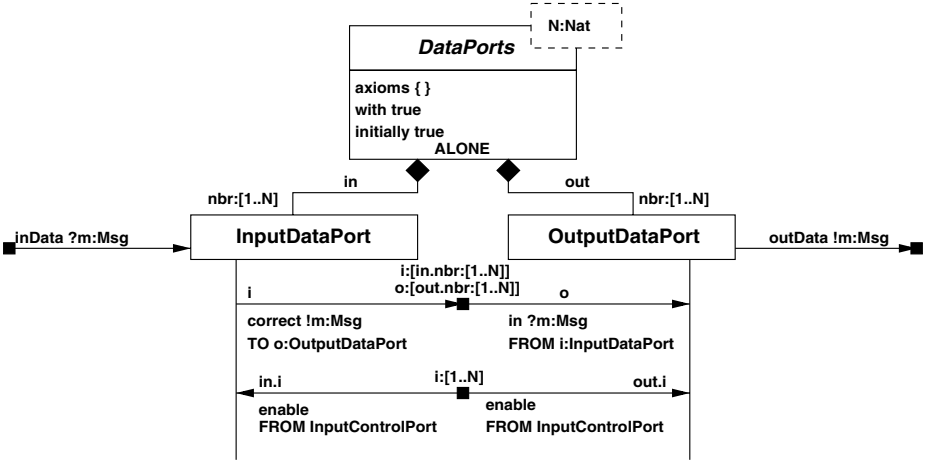

Fig. 14. `DataPorts` Communication Diagram

Each element of the transition couples (Ψ) is treated by linking the involved components with a node (boxes in Fig. 7). The parts of the couple relative to each of these components is put on the lines. Only the links between input and output ports have been represented, for example the communication between an input data port routing a correct message to the corresponding ouput data port. The elements above the links represent the participating components. Here we use again range operators as a syntactical shorthand (one link is used in place of the `N`×`N` that would be used without it):

$$\forall nbr_i \in [1..N],\ \forall nbr_o \in [1..N],\ \forall m : Msg\ .\ i = in.nbr_i,\ o = out.nbr_o \mid$$
$$\text{i.correct !m : Msg TO o} \longrightarrow\!\blacksquare\!\longrightarrow \text{ o.in ?m : Msg FROM i}$$

When components at different levels are involved, for example to treat the communication between an input control port enabling[4] both a given input data port and a given output data port, we adopt a structured communication scheme and do not add more links on the communication nodes. We add the communication information on parents (in the decomposition tree) of the concerned subcomponents, here `TransitNode` (Fig. 15). The obtaining of the KORRIGAN specification for `DataPort` from its diagram is straightforward (Fig. 16).

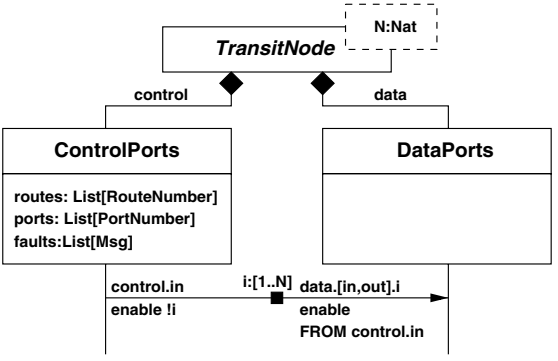

Fig. 15. `TransitNode` Communication Diagram (partial)

COMPOSITION VIEW DataPorts	
SPECIFICATION	**COMPOSITION ALONE**
imports InputDataPort, OutputDataPort **variables** N : Natural	**is** in.nbr[1..N] : InputDataPort out.nbr[1..N] : OutputDataPort **with** true,{ i:[1..N].(in.i.enable **from** s:Server, out.i.enable **from** s:Server), (i:[1..N].o:[1..N].(i.correct !m **to** o:OutputDataPort, o.correct ?m **from** i:InputDataPort), ...}

Fig. 16. `DataPorts` in KORRIGAN

Since KORRIGAN treats in a unified way both integration composition and concurrent composition, then the process and notations we have presented on

[4] This enabling scheme is used to implement the creation of object since there is no direct support for this in KORRIGAN.

concurrent communication apply on integrations too (*i.e.* for example, we would have communication diagrams between an `OutputDataPort` and its `MsgList`).

This representation of the communication is expressive enough to describe different kinds of communication, for example, both point-to-point communication (`ptp`) and broadcast communication (`broadcast`) in a client-server pattern [20]. Such a pattern may be used in the recomposition step of our method.

5 Conclusions and Related Work

We defined in previous works a formal approach based on view structures for the specification of mixed systems with both control, communications and data types. The corresponding formal language, KORRIGAN, allows one to describe systems in a structured and unifying way.

In order to make formal methods more used in the industrial world, we agree with [3]: *most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatics issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support*. Therefore, we have built a software environment, ASK [5], for the development of our KORRIGAN specifications.

In this paper, we propose a semi-formal method with guidelines for the development of mixed systems with KORRIGAN. Our method is supported by a UML-inspired graphical notation. We suggest, when possible, to reuse the UML notation, but we also present some proper extensions. Since our model is component-based, we have an approach which is rather different from UML on communications and concurrent aspects. Thus we also describe specific notations to define dynamic interfaces of components, communications patterns and concurrency. Our method is here applied to develop both textual and graphical specifications and illustrated on a transit node case-study.

Our concerns about methods and graphical notations for formal languages are close to [24,6] ones. However, we think we can reuse UML notations, or partly extend it using stereotypes, rather than defining new notations. Moreover, our approach is complementary to the theoretical approaches that try to formalize the UML. Our notations are also more expressive and abstract than [24] as far as communication issues are concerned.

KORRIGAN and UML-RT [25] partly address the same issues : architectural design, dynamic components and reusability. However, UML-RT is at the design level whereas KORRIGAN is rather concerned about (formal) specification issues. There are also some other difference, mainly at the communication level, but the major one is that, to the contrary of UML-RT, KORRIGAN provides a uniform way to specify both datatypes and behaviours.

Our notation for the glue between communicating components may be also related to [10]. The main differences are that our glue is more expressive than LOTOS synchronizations, and that we have a more structured organization of communication patterns.

We are now working on validation and verification procedures for our KORRIGAN specifications. Due to the use of STS, *i.e.* Symbolic Transition Systems, such procedures have to be adapted [15,26]. We also investigate the automatic translation of KORRIGAN specifications into PVS following the [1] methodology.

References

1. Michel Allemand. Verification of properties involving logical and physical timing features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications, ICSSEA'2000*, 2000.
2. Pascal André and Jean-Claude Royer. A First Algebraic Approach to Heterogeneous Software Systems. 14th International Workshop on Algebraic Development Techniques (WADT'99), Bonas, France, 1999.
3. M. Broy. Specification and top down design of distributed systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85*, volume 185 of *Lecture Notes in Computer Science*, pages 4–28. Springer-Verlag, 1985.
4. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.
5. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The KORRIGAN Environment. *Journal of Universal Computer Science*, 2001. Special issue: Tools for System Design and Verification, ISSN: 0948-6968. to appear.
6. Eva Coscia and Gianna Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag, 1999.
7. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
8. C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman & Hall.
9. R. France and B. Rumpe, editors. *UML'99 – The Unified Modelling Language*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
10. Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99)*, 1999.
11. Gianna Reggio and Egidio Astesiano. An extension of UML for modelling the non purely reactive behaviour of active objects. Semi-Formal and Formal Specification Techniques for Software Systems, Dagstuhl Seminar 00411, Report No. 288, H. Ehrig, G. Engels, F. Orejas, M. Wirsing, October 2000.
12. Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In Egidio Astesiano, editor, *FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.
13. David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

14. Maritta Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.
15. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
16. Bogumila Hnatkowska and Huzar Zbigniew. Extending the UML with a Multicast Synchronisation. In T. Clark, editor, *Proceedings of the third Rigorous Object-Oriented Methods Workshop (ROOM)*, BCS eWics, ISBN: 1-902505-38-7, 2000.
17. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
18. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
19. José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR'96 : Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 1996.
20. Pascal Poizat. KORRIGAN*: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*. PhD thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, December 2000. in French.
21. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.
22. G. Reggio, M. Cerioli, and E. Astesiano. An Algebraic Semantics of UML Supporting its Multiview Approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Twente Workshop on Language Technology, AMiLP 2000*, 2000.
23. G. Reggio and R. Wieringa. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In *OOPSLA'99 workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations"*, 1999.
24. Gianna Reggio and Mauro Larosa. A graphic notation for formal specifications of dynamic systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, 1997.
25. Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp., 1998.
26. Carron Shankland, Muffy Thomas, and Ed Brinksma. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag, 1997.
27. Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
28. Rational Software. Unified Modeling Language, Version 1.3. Technical report, Rational Software Corp., http://www.rational.com/uml, June 1999.
29. Kenneth J. Turner, editor. *Using Formal Description Techniques, An introduction to Estelle, LOTOS and SDL*. Wiley, 1993.
30. C. A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification Styles in Distributed Systems Design and Verification. *Theoretical Computer Science*, 89(1):179–206, 1991.